

Two-way Replacement Selection

Xavier Martinez-Palau, David Dominguez-Sal, Josep Lluís Larriba-Pey
DAMA-UPC, Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Campus Nord-UPC, 08034 Barcelona
{xmartine, ddomings, larri}@ac.upc.edu

ABSTRACT

The performance of external sorting using merge sort is highly dependent on the length of the runs generated. One of the most commonly used run generation strategies is Replacement Selection (RS) because, on average, it generates runs that are twice the size of the memory available. However, the length of the runs generated by RS is downsized for data with certain characteristics, like inputs sorted inversely with respect to the desired output order.

The goal of this paper is to propose and analyze two-way replacement selection (2WRS), which is a generalization of RS obtained by implementing two heaps instead of the single heap implemented by RS. The appropriate management of these two heaps allows generating runs larger than the memory available in a stable way, i.e. independent from the characteristics of the datasets. Depending on the changing characteristics of the input dataset, 2WRS assigns a new data record to one or the other heap, and grows or shrinks each heap, accommodating to the growing or decreasing tendency of the dataset. On average, 2WRS creates runs of at least the length generated by RS, and longer for datasets that combine increasing and decreasing data subsets. We tested both algorithms on large datasets with different characteristics and 2WRS achieves speedups at least similar to RS, and over 2.5 when RS fails to generate large runs.

Keywords

sorting, out of core sorting, heap sort, external sorting, replacement selection, merge sort, run formation

1. INTRODUCTION

Sorting is in the heart of many high performance processes [4]. Some of those processes require a sorted dataset as their final output (e.g. sort names alphabetically) or as a partial computational step (e.g. a sort merge-join). Since datasets are typically large, the selection of a good out of core sorting algorithm has an important impact on the performance of the final application. This motivates that many

benchmarks emphasize a good performance of the out of core sorting operation. For example, in the 80's Anon et al. proposed a 100MB sort benchmark, which focused on the objective to sort the dataset in the minimum time possible [1]. Even though the computer growth has outdated this particular benchmark, the sorting operation has been popular as a benchmark along the years and it is still evaluated nowadays, for instance, sorting up to 100 TB of data or sorting the maximum number of records in one minute [9].

In addition, database management systems (DBMSs) assign a memory quantum to each operation involved in a query, limiting the amount of global memory that a sort operation may use to process the whole dataset to be sorted. This raises two more issues: first, sorting becomes frequently out of core in DBMSs; and second, sorting must take advantage of the limited memory assigned by the DBMS. Out of core sorting implies that during the process, sorted runs are generated and stored in disk. It is not until all the runs are created, that they can be read again and merged to generate the final sorted dataset [4].

In summary, three features are desirable for an external sorting operation in a DBMS: (a) it should be able to start sorting data before all the input is generated; (b) it should be efficient with already sorted data because a previous operation may have already sorted or partially sorted the incoming data; and (c) the memory consumption should be predictable.

Replacement Selection (RS) has played a prominent role in these complex situations because it fulfills most of the stated features. First, unlike other sorting methods, it is able to sort data in a streamed fashion, using one heap to perform the job. Second, it generates runs which double the size of the memory available for random data, and infinite runs for already sorted data. This reduces the number of runs, allowing the merge process to reduce its fan in, and the chances to perform multiple I/O passes during the merge phase. Finally, although it is not the fastest in-memory sorting strategy, it offers a good trade off for its value: it generates smaller I/O by creating larger runs at the cost of possibly more in-core computational effort, compared to other faster in-core methods [3].

Replacement Selection, however, has one important drawback: it is not able to generate runs larger than the memory available in some cases. This creates unpredictable situations and leads to low performance in undesired cases. For example, scenarios in which the ORDER BY column is anti-correlated to the index column lead to sorting decreasing streams of data, where the performance of RS is poor. This

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... \$ 10.00.

paper solves this problem, proposing Two-way Replacement Selection, a general strategy that allows to obtain runs which are at least the size of those generated by RS and, in many cases, more than double the size of the memory available for sorting no matter the dataset, improving the good features mentioned above for RS.

Two-way replacement selection (2WRS) implements two heaps that adapt to the data characteristics, one intends to capture the growing values and the other one intends to capture the decreasing values. The strategy is to place each newly arrived record in the correct heap. But not only that, the heaps grow or shrink depending on the nature of the data. So, in case there are more growing than decreasing data, it grows the growing data heap, and shrinks the decreasing data heap, and conversely. We set two heuristics for the decision of which heap should store the new record, and grow or shrink, and we analyze them.

We also study the effect of buffering before and after the heaps without changing the total size of the memory used for 2WRS. Our study shows that the use of buffers before the insertion of data in the heaps, and the use of buffers after a decision is taken to store a data record in disk, is beneficial performance-wise. Also, we show that 2WRS scales for growing datasets, and improves RS no matter the characteristics of the dataset. The reason for the general improvement of 2WRS is the larger runs generated, and the small complexity added compared to RS.

The paper is organized as follows. In Section 2 we explain Replacement Selection. In Section 3 we introduce Two-way Replacement Selection, and then we present some theorems about the performance of 2WRS in Section 4. In Section 5 we analyze the run length obtained for different configurations of 2WRS, and in Section 6 we analyze the time performance of RS and 2WRS with the best configuration obtained from the previous section. In Section 7 we give an overview of the methods used to improve external sorting and in particular the performance of RS, and finally, in Section 8 we conclude the paper and outline some future work. We include some appendices with additional information for the paper.

2. REPLACEMENT SELECTION

Replacement Selection (RS) is an external sorting algorithm introduced by Goetz in [2]. The objective of RS is to sort a stream of records as they come (usually from secondary storage), producing another stream of released data records called “run”, which is sorted. The algorithm allocates memory that works as an intermediate buffer and stores a window of streamed data. This buffer is managed as a heap: upon the arrival of a new record, RS releases the first record in the sorting order, and stores the new record in the heap in sorted order.

This strategy has an important drawback due to the limited memory in computers. For example, when sorting in ascending order, any new record introduced in the heap that is smaller than the last flushed record to the stream of released records cannot be included in the active run: it would not be possible to place such record among the already flushed records. These records are kept at the bottom of the heap, marked as “next run” records, until they fill all the heap. Note that these marked records preserve the heap structure among them.

When the algorithm closes the current run, it starts a new

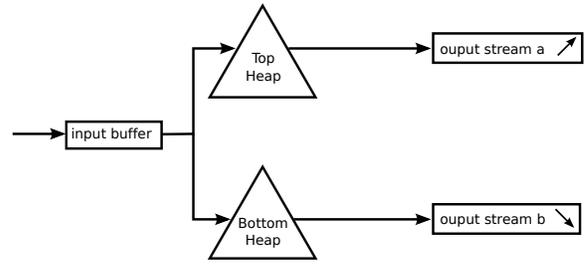


Figure 1: Functional diagram of the simplified version of 2WRS.

run which contains all the records that could not be included in the closed run. This breaks the incoming flow of data into multiple runs that are merged in a final phase to create a final single run. Appendix A presents the pseudocode of the main loop of RS.

The merge phase is strongly dependent on how many runs have been generated before. Runs are stored sequentially in disk, and in order to improve the merging speed it is desirable to have the longest possible runs. Thus, the ideal situation would be when a single run is generated, which, for instance, occurs when the input is already sorted.

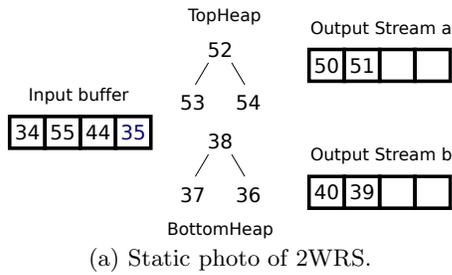
The theoretical analysis of RS found that, for inputs following a uniform random distribution, the average run is twice the size of the available memory [4].

3. 2-WAY REPLACEMENT SELECTION

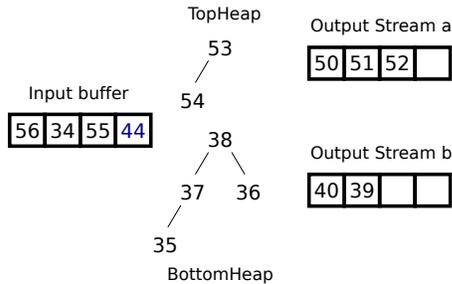
The objective of this paper is to get the most out of RS, i.e. generate runs above double the size of the memory available, no matter the incoming dataset characteristics. This is achieved with two-way replacement selection which is explained below.

Two-way replacement selection (2WRS) implements two heaps that we call TopHeap and BottomHeap, instead of one as in RS. The objective of the two heaps is that they cooperate to obtain longer runs: the TopHeap and BottomHeap capture increasing and decreasing sequences of values respectively. This architecture resembles two cooperating RS algorithms working together, which output their result into streams *a* and *b*, as depicted in Figure 1. Therefore, stream *a* is a sequence of increasing values and stream *b* is a sequence of decreasing values that do not overlap.

In order to decide which heap should be populated with every new input record, we take a decision based on an input buffer and a heuristic. The input buffer is an array of records that acts as a regular I/O buffer, which loads data items from disk and releases them with a FIFO strategy, as shown in Figure 1. When a new record has to be inserted in one of the heaps, 2WRS chooses randomly one of the root records of the heaps, which is released to the corresponding stream, *a* or *b*. In an independent decision, the ‘input’ heuristic samples the input buffer and takes a decision on which heap, either the TopHeap or the BottomHeap, will store the record obtained from the FIFO structure. If the heuristic chooses the heap that released the top record, then the two heaps preserve their size. Otherwise, one heap grows and the other one shrinks. The process is iterated until no more records can be removed from any of the heaps (because all the records are marked as belonging to the next run) and then the whole algorithm is restarted to process a new run.



(a) Static photo of 2WRS.



(b) Record 35 has been processed.

Figure 2: Example of the simplified 2WRS algorithm.

We propose and compare the following two input heuristics (see Appendix B for more proposals on heuristics):

Random Every record is pushed into a heap selected at random. With this heuristic, the expected size of the heaps is the same.

Mean If the record to be pushed is larger than the mean of the records in the buffer when the decision is taken, then it is inserted into the TopHeap. Otherwise, it is inserted into the BottomHeap. This heuristic captures a rough approximation of the data distribution and balances the heaps according to the distribution.

Example. The example in Figure 2(a) shows a static photo of the process after a few records have been inserted into streams *a* and *b* using the *Mean* input heuristic and two heaps of size three records, i.e., six records in total. The input data for all the examples in this section is

{40, 50, 39, 51, 38, 52, 37, 53, 36, 54, 35, 44, 55, 34, 56, ...}

As we can see in Figure 2(a), stream *a* that is generated from the TopHeap, starts with value 50 and receives growing records. Stream *b* is generated from the BottomHeap, starts with value 40 and receives decreasing records. During the merge process, the run is formed reading stream *b* backwards and then stream *a* forwards. Stream *b* has records between the minimum value and 40, and stream *a* has records between 50 and the maximum value.

Let us see where this situation comes from. The input heuristic plays an important role in how the heaps are filled at the beginning of each run. So, in this case, the first four records, which are {40, 50, 39, 51}, fill the input buffer and the heuristic decides which heap is going to store the first record, 40. If we choose the *Mean* input heuristic, the mean of the four values is 45. Given that 40 is below the mean, the BottomHeap will store it. The heuristic is used while

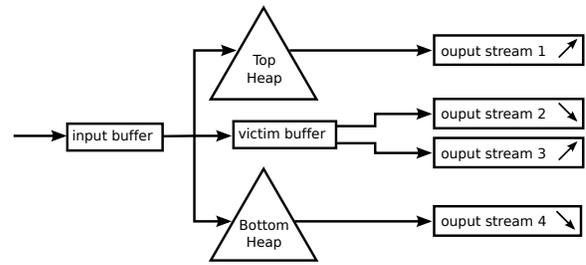


Figure 3: Functional diagram of 2WRS.

the two heaps are not full, or when they are being filled with next run records.

Let us go back to the example in Figure 2(a), and see how the next record in the input buffer, i.e., 35, would be processed. This is the case where the input heuristic is not necessary. First, we decide at random which of the top records of the two heaps will be written to the corresponding stream. In this case, let us suppose that we decide randomly for record 52 of the TopHeap. Now, record 35 can only be stored in the BottomHeap, which increases in size, stealing one position from the TopHeap. This is shown in Figure 2(b).

The next record, 44, belongs to the next run because it is between 38 and 53, which are the top values of the BottomHeap and the TopHeap, respectively.

Victim Buffer

Note that between streams *a* and *b*, there is a gap that, for the example in Figure 2, is between 40 and 50. This gap is fixed at the very moment that streams *a* and *b* are created. Thus, it gives room for creating a strategy to sort records that fit there. This is the objective of the victim buffer.

Figure 3 shows the functional diagram of 2WRS with the input buffer, the victim buffer, and the four streams necessary. Note that streams *a* and *b* in the previous explanation change into streams 1 and 4 here. Also, the victim buffer will manage two streams, 2 and 3, with the help of a heuristic.

At the beginning of the generation of each run, the victim buffer is empty and has to be initialized. At this stage, the records that are removed from the TopHeap and the BottomHeap (which would normally be written to streams 1 or 4) are stored in the victim buffer. When the victim buffer is filled with records, its contents are sorted. Once sorted, each two consecutive records of the victim buffer define a range. The largest of these ranges is selected as the victim buffer valid range. The sorted records of the victim buffer are flushed to streams 2 and 3. The smallest of the two records that define the new valid range, along with those smaller than it are flushed into stream 3 and the rest into stream 2. At this point, the initialization phase ends.

Every time the victim buffer is full, it is sorted and flushed according to the new valid range, which is selected as in the initialization phase. Again, victim buffer records smaller than the minimum of the new valid range are flushed to stream 3 and the rest to stream 2. This assures that records in stream 3 are always smaller than records in stream 2, which preserves the nature of 2WRS.

Now, the procedure for any record coming from the input buffer is similar to that before, but introducing the victim buffer. However, if the record is out of the range of streams 1

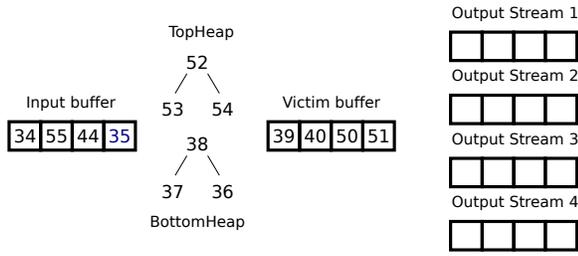


Figure 4: Example of 2WRS.

and 4, we first have to check if it can be stored in the victim buffer, or it belongs to the next run.

Appendix A includes the pseudocode of the main loop of 2WRS. This algorithm includes the whole process with the input buffer, the victim buffer and the four streams.

Example. Figure 4 shows the initialization phase of the victim buffer after the first four records have already been sorted. Now, records 40 and 39 will be written into stream 3, and records 50 and 51 into stream 2.

Now, the situation for record 44 is similar to the previous example. However, given that it is between 40 and 50, which is the valid range of the victim buffer, it is stored in the victim buffer and belongs to the current run. Note that thanks to the victim buffer, the algorithm is able to generate a longer run in this case.

Discussion. As shown in Figure 3, streams 2 and 3 will grow until the largest and smallest values in the victim buffer do not fit any input record between them. We note that after the initialization phase the elements removed from the TopHeap and BottomHeap are written into streams 1 and 4, respectively ¹.

The algorithm creates its result in four different output streams, as opposed to RS, which only uses one stream. However, those four streams are consecutive and non overlapping among them: (1) streams 1 and 3 are sorted in ascending order, (2) streams 2 and 4 are sorted in descending order; (3) any four records x, y, z, w from streams 1, 2, 3 and 4 respectively hold: $x \leq y \leq z \leq w$. Therefore, it is immediate to generate the final run as the sequence of the data generated by streams 4, 3, 2 and 1 in this order. Appendix C details how each stream is stored in disk and how to use only four files to store each run.

Among other situations, the use of streams 2 and 3 and the victim buffer are very beneficial for convergent series, while stream 1 captures increasing series, and stream 4 decreasing ones. The output streams 1 and 4 define an interval of values (those which fall among both streams) that cannot

¹It would be possible to start filling the victim buffer before the two heaps are completely filled. However, the selection of victims is improved if we start filling the heaps because more information from the input is available: the objective is to select the widest gap. We load in memory as much data possible and then, the victim buffer is filled with those records that maximize the gap. Also, the victim buffer may be simplified by using only one stream instead of two. But with this variant, it is impossible to select a new range of records for the victim buffer after each flush, which makes the variant unsuitable, for instance, for convergent series. With the addition of the second stream, any range can be selected between those defined by two consecutive records in the victim buffer and, in particular, we select the largest one.

be released in the current run. Therefore, the use of the victim buffer captures this trend.

2WRS behaves identically when the input is already sorted as well as when it is sorted in reverse order because it takes advantage of the TopHeap or the BottomHeap, respectively. In both cases, the runs are of infinite size. Furthermore, the presence of the victim buffer also allows 2WRS to generate runs of infinite size for convergent series, which contain sequences of values that keep approximating. In summary, 2WRS is able to detect structured increasing or decreasing inputs and benefits from their regularity to build longer runs.

4. ANALYSIS

In this section we present six theorems which describe some properties of RS and 2WRS. Theorems about RS properties are presented for comparison purposes (Theorems 1, 3 and 5). We show that 2WRS is able to sort incoming data, generating runs which are, at least as long as those generated by RS for random data (Theorem 2) and longer than RS for other data inputs (Theorems 4 and 6). Proofs of these theorems can be found in Appendix D.

THEOREM 1. *For inputs already sorted in ascending order, RS generates one run with all the input records.*

THEOREM 2. *For inputs already sorted in ascending order, 2WRS generates one run with all the input records.*

THEOREM 3. *For inputs sorted in reverse order, RS generates runs with length equal to the size of the memory.*

THEOREM 4. *For inputs sorted in reverse order, 2WRS generates one run containing all the input records.*

THEOREM 5. *For inputs consisting of alternating chunks of length k records sorted in ascending order and k records sorted in descending order repeatedly, RS generates runs with an average length around twice the size of the memory m ($m \ll k$).*

THEOREM 6. *For the inputs of Theorem 5, two-way replacement selection generates runs with an average length equal to k (with an appropriate heuristic²).*

5. RUN LENGTH ANALYSIS

In this section, we analyze the configuration parameters of 2WRS following the analysis of variance (ANOVA). The ANOVA detects which variables are more relevant and it is used to select the optimal configuration for a set of variables or factors (for further details about ANOVA, see [10]). Our output variable will be the length of the runs, and hence, the variable to be optimized. In these experiments, the memory size allocated to the algorithm is fixed to 100k records and the input length is 1GB. Each record is formed by a 4B integer.

The observations are obtained as a crossed factorial experiment with four variables or factors:

- *Buffer setup:* We test three configurations: only input buffer, only victim buffer, and both input and victim buffer.

²An appropriate heuristic is one that uses the TopHeap for sorted inputs and the BottomHeap for reverse sorted inputs.

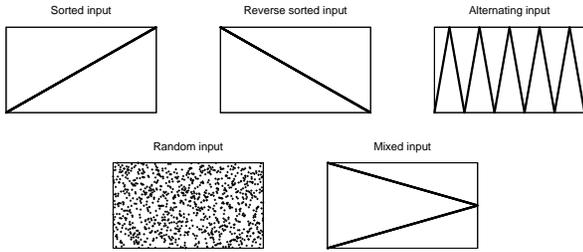


Figure 5: Samples of all data inputs used.

- *Size of buffer:* We set three configurations: 0.2%, 2% and 20% of the available memory are dedicated to the buffers and the rest to the heaps. Note that in all the configurations, the total allocated memory (the addition of the heap and buffer sizes) for 2WRS is always constant.
- *Heuristic:* We test two configurations for the heuristic of the input buffer: random and mean.
- *Data distribution:* We test five different data input distributions.
 1. Sorted: the records are already sorted.
 2. Reverse sorted: the inputs are sorted in reverse order.
 3. Alternating: this dataset is a sequence of one increasing interval followed by one decreasing interval. The number of intervals is set to 50, with 25 increasing and 25 decreasing interleaved intervals.
 4. Random: the records are generated following a uniformly random distribution.
 5. Mixed: this dataset alternates one record from a sequence of increasing records, with another record of a sequence of decreasing records.

We depict these datasets in Figure 5.

In order to add some randomness to the experiments a uniformly distributed random value is added to each input. These random values range from 1 to 1000 for a total range of values sorted from 1 to 10^9 .

Different combinations of these input datasets build up more complex data distributions that can be found in several scenarios. For example, if the execution plan of an SQL query performs two merge joins of two attributes in a table that are correlated inversely, then the first join reads the data in one order, and the second join reads it in reverse order. Also, the mixed dataset may be the input to a sort operation that reads a union of two merge joins: since the execution might be pipelined and parallel, the input to the sort might mix increasing and decreasing streams.

In Table 1, we summarize the average run length for all the input sets. In this table, we show the run length of RS compared to the best three parameterizations of 2WRS, which all use the *Mean* input heuristic. Cfg 1 maximizes the run length for the random dataset, uses 0.2% of memory for buffers and no victim buffer. Cfg 2 maximizes the run length for mixed dataset, uses 20% of memory for both buffers. Cfg 3 works reasonably well for all inputs, and uses 2% of memory for both buffers.

Input	RS	2WRS cfg 1	2WRS cfg 2	2WRS cfg 3
Sorted	inf	inf	inf	inf
Reverse sorted	1.0	inf	inf	inf
Alternating	1.94	50	50	50
Random	2.0	2.0	1.6	1.96
Mixed	2.0	1.2	16.5	2.24

Table 1: Average run length relative to memory size.

For the remaining data inputs (sorted, reversed and alternating), we have found that all configurations of 2WRS are optimal: 2WRS generates runs of infinite size for the sorted and the reverse sorted datasets, and 2WRS builds runs of length 50 times the memory size for alternating sequences of length 50. On the other hand, RS is only able to generate runs of size equal to the memory available for the reverse sorted and equal to approximately twice the memory for the alternating data, as found in Theorems 5 and 6. This shows that when datasets are sorted or partially sorted 2WRS is more effective than RS.

The configuration parameters, and specially the buffers and the heuristic, have an important effect on the mixed datasets. Although not shown in Table 1, the configurations without both the input and victim buffers show poor run lengths. Additionally, we computed the average run length with respect to the heuristic used, which were 3.0 for mean and 1.85 for random. A t-student test confirmed that with a significance level of 0.05, these two averages are different. Therefore, we conclude that the mean heuristic is better than the random heuristic.

Also in Table 1 we see that Cfg 1, that does not use the victim buffer, improves the performance of RS with reverse sorted and alternating inputs. Cfg 2 and Cfg 3, that use the victim buffer, work better than Cfg 1 with the mixed input. Thus, the addition of the victim buffer improves the performance for input data distributions that show convergent trends.

With respect to the random distribution, we found that 2WRS is as good as RS because none of them is able to take advantage of any pattern in the distribution. In our results, we observed that for random distributions, there is a linear correlation between the buffer size and the run length. If buffers are allocated, the memory dedicated to the heap diminishes by this percentage. Thus, a configuration with 2% of the memory dedicated to buffers, reduces the run length by just 2% for random distributions, as shown in Figure 6. Furthermore, in our experiments, we measured a very small difference in the length of the runs generated by the configurations with 0.2% and 2% allocated to the buffers, but larger between 2% and 20%.

All in all, our run length analysis concludes that 2WRS creates runs of a length at least very close to RS or significantly better. 2WRS is able to capture partially sorted data such as those in the alternating and mixed datasets, and is optimal with totally sorted data either increasingly or decreasingly. Regarding the configuration, we found that the presence of buffers and the mean heuristic is very important because they generate longer runs for the datasets with more complex structures. Also, the use of small buffers is sufficient to obtain statistically significant larger runs with 2WRS than with RS.

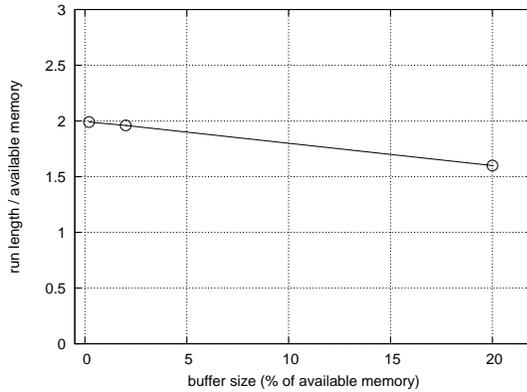


Figure 6: Length of runs relative to memory size as a function of buffer size for random inputs.

6. PERFORMANCE ANALYSIS

In this section, we test experimentally the performance of 2WRS with respect to RS. In all our experiments, we account for the time to generate the runs, as well as the subsequent merge phase. We generate different datasets following the random, mixed, alternating and decreasing patterns and we measure the sorting time for each strategy. We do not show the results for the sorted dataset because RS and 2WRS are equivalent.

All the 2WRS configurations used in the experiments analyzed in this section use the mean heuristic, since this combination generates longer runs overall for all the inputs analyzed as we have seen in Section 5. According to the previous section, a large buffer benefits the sorting of mixed datasets, and a tiny buffer benefits random inputs. Therefore, we set the available memory for the buffers to an intermediate value, which is 2%.

We perform two experiments varying the input length and the memory allocated to the sort algorithm. In the first experiment, we fix the input to 256k records (1GB of data) and vary the memory to fit from 1k to 1M records. Therefore, our tests aim at systems with inputs larger than the memory available (between 3 and 6 orders of magnitude for the examples). In the second experiment, we fix the memory to fit 10k records and we vary the input from 25M to 256M records (100MB to 1GB).

Setup: We execute the algorithms in a computer equipped with an Intel Core 2 Duo processor running at 2.40 GHz. Each core has 4 MB of L2 cache memory and the system has a total of 2 GB of RAM. The hard disk is a SATA drive with a capacity of 60 GB. The OS of the system is Debian GNU/Linux. Given that we want to limit the available memory dedicated to sorting, we open all files using direct I/O, which bypasses the operating system cache.

Fan in analysis: The merge phase is computed as a tree of run merges. Depending on the number of files merged simultaneously (i.e. the fan in), the performance of the algorithm varies. In an experiment detailed in Appendix E, we measured the best fan in for merging runs in our experimental setup, and we obtained that the optimal fan in is equal to 10 runs. Thus, in all the following experiments with RS and 2WRS, we use this fan in for the merge phase.

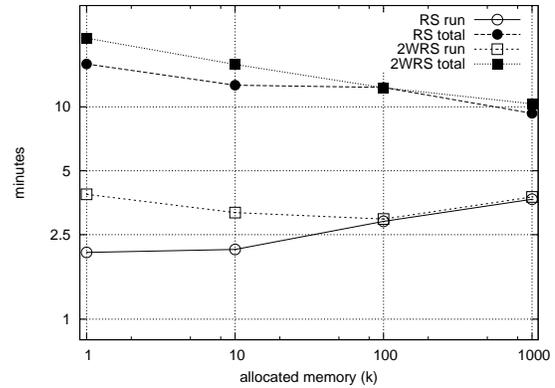


Figure 7: Run generation and total sorting times for random input as a function of the available memory.

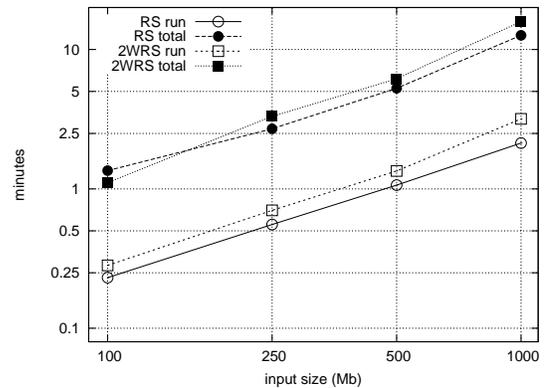


Figure 8: Run generation and total sorting times for random input as a function of the input size.

6.1 Random

In Figure 7, we plot the performance with respect to the memory buffer size. The time spent generating the runs is detailed with empty circles and squares for RS and 2WRS respectively, and the total time needed to sort is depicted with solid circles and squares. The same applies to the following plots. We observe that the total time needed by the two algorithms is very similar. This is due to the fact that it is not possible to predict the behavior of random input data. 2WRS has slightly worse performance during the run building phase for some configurations because the logic of 2WRS is slightly more complex than for RS, due to the two heaps and the multiple streams. However, the difference between both algorithms is tiny, and thus the use of either RS or 2WRS is similar for random inputs.

We plot the scalability of the algorithms with respect to the input length in Figure 8. Here, we observe a similar pattern to that described for the previous plot, where 2WRS is only 10% slower than RS due to the introduction of buffers and their management. Furthermore, we observe that both algorithms scale identically when the input size grows.

6.2 Mixed

In the run length section, we found that 2WRS creates runs significantly larger than RS for mixed datasets. Fig-

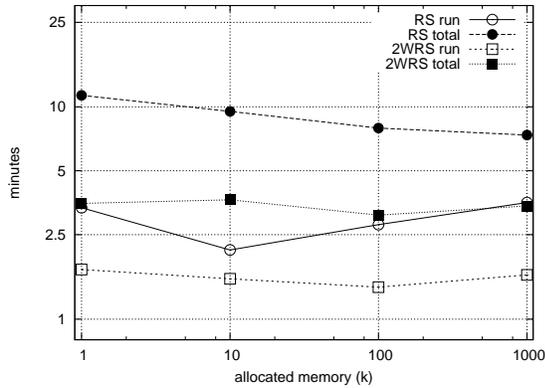


Figure 9: Run generation and total sorting times for mixed input as a function of the available memory.

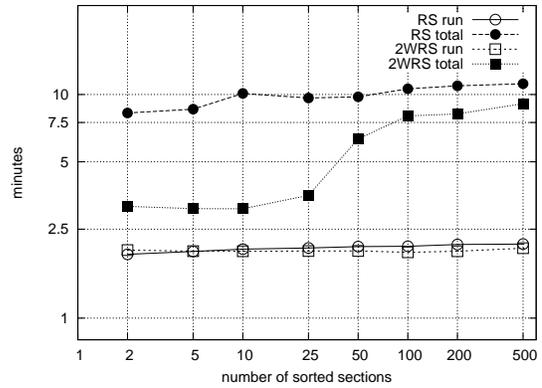


Figure 11: Run generation and total sorting times for alternating input as a function of the number of sorted and reverse sorted sections.

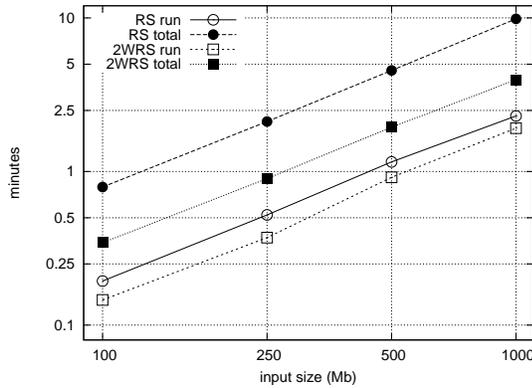


Figure 10: Run generation and total sorting times for mixed input as a function of the input size.

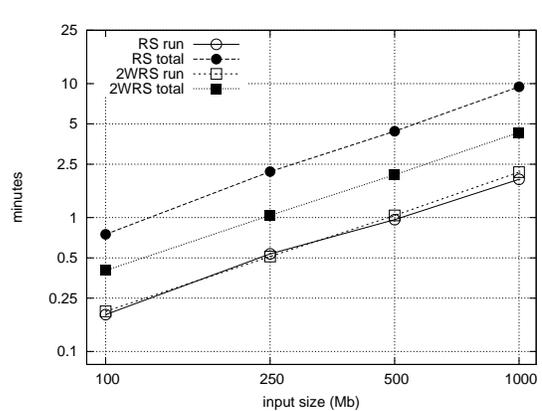


Figure 12: Run generation and total sorting times for reverse sorted input as a function of input size.

Figure 9 confirms this, because independently of the memory size, 2WRS is approximately three times faster than RS. This is because 2WRS generates less runs for the mixed dataset, and so the merge phase is much faster than with RS, which is not shown explicitly in the plot. We also see that, as the amount of allocated memory increases, both algorithms need less time to sort the data, since the runs generated are longer, and thus less merge phases are needed.

In Figure 10, we represent the scalability of both algorithms with the input. The advantage of 2WRS over RS for mixed data is sustained as the input data grows, and for all input sizes an approximate speedup of 3 is maintained. We note that for this dataset even the run generation of 2WRS is faster. This is because the heaps are not used and most of the computational time is spent sorting the victim buffer. Since the victim buffer uses a standard library sort, which is optimized for efficient in memory sorting, it is faster than RS that applies a heapsort.

6.3 Alternating

The complexity to sort the alternating dataset is dependent on the number of increasing and decreasing intervals for a fixed input size. If there are very few intervals, the dataset is similar to the sorted dataset, but if there are many intervals then it becomes closer to the random dataset. In this

experiment, we fix the memory allocated to the algorithms to 10k and the input size to 256k records, and we vary the number of increasing and decreasing sections. In Figure 11, we depict the sorting time for both algorithms.

For a small number of sorted sections, 2WRS performs much better than RS, achieving up to an approximate speedup of 3. We observe that although the run phase takes the same time for both algorithms, the merge phase is significantly shorter for 2WRS because of the fewer number of runs that are generated. 2WRS is able to include the sections sorted in reverse order in a single run, whereas RS creates multiple runs for these sections. As the number of peaks increases, the sorted sections are shorter. Then, both algorithms asymptotically tend to need the same amount of time for sorting the data, although 2WRS still performs better, because it still separates ascending and descending intervals. In the extreme case, if the number of peaks tends to infinite, the dataset would resemble a random input and both algorithms would spend the same execution time for the purpose.

6.4 Reverse Sorted

In Figure 12, we plot the time spent by both algorithms to order reverse sorted data, as the size of the input grows. We

observe that for all input sizes 2WRS gets a better performance than RS. The scalability of both algorithms is similar, showing parallel trends, that indicate a constant speedup, which is in this case 2.5.

7. RELATED WORK

Sorting is a basic computing problem that has received a lot of attention over the years. The basis of most external sorting algorithms is a two step process that in the first phase generates runs as long as possible, and in the second phase it merges the runs. Often, the run generation phase is based on some internal sorting algorithm. In particular, replacement selection is based on heapsort, which is analyzed in [4].

In 1998, Larson and Graefe experimentally compared different memory management algorithms during run generation when ordering variable length inputs, showing that replacement selection is a viable algorithm for commercial database systems [7]. Moreover, a recent survey on sorting in database systems pointed out that replacement selection is one of the most used techniques for external sorting in databases [3].

Replacement selection was introduced by Goetz in [2] and since then several modifications and alternatives have been proposed. For instance, Larson introduced a modified version of RS called batched replacement selection, a cache conscious version that also works for variable length records [6]. More recently, Koltsidas, Müller and Viglas introduced a new variation of replacement selection for sorting hierarchical data (e.g. XML files) [5].

There have been several proposals to improve the performance of the merge phase, but no emphasis has been placed on the generation of larger runs in the general case. Zheng and Larson introduced a new reading strategy for external mergesort that consistently performs better than double buffering and forecasting [12]. This technique uses heuristics to precompute the order in which data blocks will be read during the merge phase.

Yiannis and Zobel studied the possibility of compressing sets of records during the run generation phase in order to reduce disk and transfer costs of external sorting by reducing the number of runs generated, and proposed a new compression technique adapted to sets of records [11].

We should note that all modifications and improvements of RS can be readily applied to 2WRS without change, so 2WRS also benefits from all these changes.

8. CONCLUSIONS

In this paper, we propose Two-way Replacement Selection (2WRS), which is a generalization of Replacement Selection (RS). 2WRS allows to deal with increasing, decreasing and mixed inputs, obtaining runs of optimal size, and significantly longer than RS. Moreover, this improvement does not penalize significantly the length of the runs of 2WRS for random distributions, the length of which is similar to those of RS. Besides, the additional complexity generating the runs is amortized by a faster merge phase, which turns into a much faster total execution time.

Our results have been tested for different input sizes and space dedicated to the sorting operation. We have been able to sort datasets with strong memory limitations (6 orders of magnitude larger) three times faster than the regular

RS. Furthermore, we obtain similar speedups with different scaleups of the input.

Additionally, the amount of memory allocated to 2WRS can be fixed beforehand as with RS, which makes our proposal also suitable for DBMSs. Finally, 2WRS maintains the heap and run generation architecture of RS that allows for improvements already proposed in the literature for RS, which include variable key support, read ahead strategies or hierarchical data sorting among others.

9. ACKNOWLEDGMENTS

The authors thank the Ministry of Science and Innovation of Spain and Generalitat de Catalunya, for grant numbers TIN2009-14560-C03-03 and SGR-1187 respectively.

The authors want to thank the anonymous reviewers for their useful and insightful comments.

10. REFERENCES

- [1] Anon et al. A measure of transaction processing power. *Datamation*, 31:112–118, 1985.
- [2] M. Goetz. Internal and tape sorting using the replacement-selection technique. *Communications of the ACM*, 6(5):201–206, 1963.
- [3] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3), 2006.
- [4] D. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, 2nd edition, 1998.
- [5] I. Koltsidas, H. Müller, and S. Viglas. Sorting hierarchical data in external memory for archiving. *Proceedings of the VLDB Endowment archive*, 1(1):1205–1216, 2008.
- [6] P. Larson. External sorting: run formation revisited. *IEEE TKDE*, 15(4):961–972, 2003.
- [7] P. Larson and G. Graefe. Memory management during Run Generation in external Sorting. In *SIGMOD*, pages 472–483. ACM, 1998.
- [8] X. Martinez-Palau, D. Dominguez-Sal, and J. L. Larriba-Pey. Two-way replacement selection. Technical Report UPC-DAC-RR-DAMA-2010-1, 2010.
- [9] Minute sort web page. <http://sortbenchmark.org>.
- [10] D. Montgomery. *Design and Analysis of Experiments*. Wiley, 5th edition edition, 2000.
- [11] J. Yiannis and J. Zobel. Compression techniques for fast external sorting. *The VLDB Journal*, 16(2):269–291, 2007.
- [12] L. Zheng and P. Larson. Speeding up external mergesort. *IEEE TKDE*, 8(2):322, 1996.

APPENDIX

A. PSEUDOCODE

In this appendix we include the pseudocode of the main loop of the Replacement Selection and Two-way Replacement Selection algorithms.

A.1 Replacement Selection

Algorithm 1 shows the pseudocode for the main loop of RS. In the first phase, method *heap.fill* loads the first records from the input into the heap.

Algorithm 1 RS(heapSize)

Require: The maximum size of the heap $heapSize$.

Ensure: Each run is sorted.

```
1: let current a pair of integers containing a value for a
   record and the run to which it belongs.
2: let heap a minheap, of maximum size heapSize.
3: let currentRun an integer.
4: let nextOutput an integer.
5: heap.fill(inputBuffer);
6: currentRun = 0;
7: while heap.size() > 0 do
8:   nextOutput = heap.pop()
9:   output(nextOutput);
10:  //Read next value
11:  if input.read(current.value) then
12:    if current.value < nextOutput then
13:      current.run = currentRun + 1;
14:    else
15:      current.run = currentRun;
16:    end if
17:    heap.insert(current);
18:  end if
19:  //Start next run?
20:  if heap.top().run > currentRun then
21:    currentRun = 1 + currentRun;
22:  end if
23: end while
```

Then, the main loop is executed while the heap is not empty. First, a record is output to make room for a new one. This is done by method *output*. Next, a record is read from the input. If the record is smaller than the last output record, it is marked as belonging to the next run, else it is marked as belonging to the current run. Next, the top record of the heap is removed with *heap.pop* and the record read from the input is inserted in the buffer with *heap.insert(current)*.

Finally, when the top record of the heap belongs to the next run, i.e. it is too small to be part of the current run, the current run ends. The process starts again, and a new run is generated until the input is fully read.

A.2 Two-way Replacement Selection

Algorithm 2 shows the pseudocode for the main loop of 2WRS. The algorithm, first fills both heaps with records obtained from the input. This is done by method *doubleHeap.fill*. When a record can be stored in both heaps, this function uses the heuristic to decide which heap is used to store the record.

The main loop is executed while the two heaps are not empty. First, a record is released to make room for a new one. This is done by method *victimBuffer.output*, which pops the top record from either the TopHeap or BottomHeap at random.

Next, a record is obtained from the input buffer. Method *victimBuffer.fit* checks whether the current record is inside the gap currently processed by the victim buffer and, if so, stores it and returns *true*. Otherwise, it does nothing and returns *false*. Note that at the beginning of each run, while the victim buffer has not been completely filled, this function always returns *false*. While this method returns *true*, new

Algorithm 2 2WRS(inputBuffer, heapSize, victimBufferSize, inputHeuristic)

Require: An input buffer *inputBuffer*, the maximum combined size of the heaps *heapSize*, the victim buffer size *victimBufferSize* and the input heuristic *inputHeuristic*.

Ensure: The generation of several ordered runs.

```
1: let current a pair of integers containing a value for a
   record and the run to which it belongs.
2: let doubleHeap a pair of heaps, a maxheap and a min-
   heap, of maximum total size heapSize.
3: let victimBuffer a victim buffer of size victimBufferSize.
4: let currentRun an integer.
5: doubleHeap.fill(inputBuffer, inputHeuristic);
6: currentRun = 0;
7: while doubleHeap.size() > 0 do
8:   output(doubleHeap);
9:   if inputBuffer.read(current.value) then
10:    current.run = currentRun;
11:    while victimBuffer.fit(current.value) do
12:      inputBuffer.read(current.value);
13:    end while
14:    doubleHeap.insert(current);
15:  end if
16:  if doubleHeap.nextRun(currentRun) then
17:    currentRun = 1 + currentRun;
18:    victimBuffer.flush();
19:  end if
20: end while
```

records are read from the input buffer. When the record read can not be put in the victim buffer, the method returns *false* and the record is inserted into one heap by method *doubleHeap.insert*. This method inserts the record into one of the heaps, using the first heuristic when necessary.

Finally, method *doubleHeap.nextRun* returns *true* when the top record of both heaps belong to the next run, meaning that the current run reached an end, since all records in memory also belong to the next run. In this case the records stored in the victim buffer are written to disk and the next run starts, with an empty victim buffer.

The main loop of this algorithm is very similar to the main loop of replacement selection, shown in the previous subsection. The only difference is that 2WRS checks in the main loop whether the current record can be placed in the victim buffer and keeps reading new records while this is the case.

B. HEURISTIC ANALYSIS

In addition to the analysis presented in Section 5, a more extensive analysis of variance is available in [8], which is summarized in this Appendix.

Apart from the *Random* and *Mean* heuristics, we also tested four other heuristics:

- *Alternate*: records are assigned to the BottomHeap and TopHeap alternatively. If a record is inserted in the BottomHeap, the next one will be inserted in the TopHeap, and vice versa.
- *Median*: behaves similarly to the *Mean*, but the next record is compared with the median of the records.
- *Useful*: keeps a dynamic track of the usefulness of each

heap. The usefulness of a heap is measured as the number of records output by that heap divided by its size. New records are stored in the most useful heap.

- *Balancing*: records are stored in the smallest of the two heaps. When a run starts, if one heap has more records than the other, records are popped from the large heap and inserted into the small one until both heaps contain the same number of records.

Additionally, the algorithm description states that when a record has to be output, it is selected at random between the top records of both heaps. This selection does not need to be random, and it is referred to as output heuristic. We tested four other output heuristics in addition to *Random*:

- *Alternate*: chooses the heaps in an alternating fashion. First, a record is popped from the BottomHeap, and the next one from the TopHeap.
- *Useful*: using the same usefulness measure as the *Useful* input heuristic, the record is popped from the most useful heap.
- *Balancing*: keeps both buffers the same size. Thus, the record is popped from the larger heap.
- *Min distance*: the first output is chosen at random, and for the following ones, the closer record in absolute value to the first output is selected.

Note that the introduction of this heuristic in the pseudocode of Appendix A does not introduce significant changes but in method *output* where a new parameter has to be added, which is this heuristic.

For both types of heuristics, *Random* is considered the baseline, as we consider it to be the simplest solution. The different ANOVA models analyzed show that the best input heuristics are the *Mean* and *Median*. *Mean* is selected over *Median* because its implementation is more efficient. As for the output heuristic, the analysis shows that the *Random* and *Balancing* output heuristics have the best performance. However, there is not enough evidence to assert that *Balancing* is better than *Random* or vice versa.

In order to keep the presentation of the algorithm clear and simple, the description of the output heuristics is omitted because none is found to be better than *Random*, and only the *Random* and *Mean* input heuristics are described because they are the baseline and the best input heuristic, respectively.

C. STORING DECREASING RECORDS

Due to the way 2WRS works, it generates two streams of sorted records (streams 1 and 3) and two streams of reverse sorted records (streams 2 and 4). The latter need to be stored already sorted in order to allow the merge phase to read files sequentially.

In order to store the data supplied by a stream while reversing the order of the records, a file is created with a fixed size of k pages. The data records read from the stream are written to the file starting at the end, that is, the last position of the last page, and continuing backwards until the first page is reached. When the file is full, a new one is created in the same manner.

In order to minimize the number of input/output operations with the hard disk, a special output buffer is used with each file. This buffer has the same size as a disk page. When

a record from a stream is output, it is written to this buffer instead of being immediately written to the file. When the buffer is full, its contents are flushed to the corresponding page of the file. The memory space needed to store the buffer is taken from the memory dedicated to the 2WRS algorithm, but the performance of the algorithm is not affected because, typically, the amount of memory available to a sorting algorithm is several orders of magnitude larger than the size of a file page.

The first page of each file contains a header with the following information:

- *Number of files*: the number of files that have been created by the corresponding stream in the current run.
- *Number of pages*: the number of pages that each file has.
- *Offset*: the page and the position within that page where the data begins. This should be page number two and first position for all files except possibly the last one.

In order to know the order in which the files have been created, we use a naming system that assigns each file the same name followed by a different number. With this naming system, it is possible to open directly any of the files created to store the stream, and read the whole sequence in the desired order.

Once all the files have been created, the records can be read in non decreasing order starting from the last file created and ending at the first one.

The value of k should be chosen large enough so that not a large number of files are created, since closing and opening files adds an unnecessary time overhead. But if the value of k is chosen to be very large, it is possible that most runs fill only a small portion of the file, and lots of hard disk space will be wasted containing huge temporary files. Thus, when deciding the value of k , one must consider the number of records to be sorted, the expected run length, and the hard disk space available. This value can also be adapted during the execution of the 2WRS algorithm and it can change between runs. We use a value of $k = 1000$ to ensure that few files are created each run, which corresponds to 40MB files.

In summary, this storage policy allows to build a run for 2WRS simply concatenating the files generated by streams 4, 3, 2 and 1 for the current run.

D. ANALYSIS

In this Appendix we present the proofs of the theorems in Section 4.

THEOREM 1. *For inputs already sorted in ascending order, RS generates one run with all the input records.*

PROOF. Since the input records are already sorted, each new record will be larger than all the values in the heap and, thus, it will be possible to insert it into the heap as belonging to the present run. No record will be marked as belonging to the next run. \square

THEOREM 2. *For inputs already sorted in ascending order, 2WRS generates one run with all the input records.*

PROOF. The same proof as for Theorem 1. All records obtained from the input are larger than those stored in memory. All the records are stored in the TopHeap, and all they belong to the same run. \square

THEOREM 3. *For inputs sorted in reverse order, RS generates runs with length equal to the size of the memory.*

PROOF. Since the input records are sorted in reverse order, the next record obtained from the input is smaller than all the previous records. Thus, it is not possible to include the new record in the current run when the heap is full. So, the new record is marked as belonging to the next run. When the heap is full every new record belongs to the next run. Once the records belonging to the present run are released, a new run starts and the size of the run is equal to the available memory. \square

THEOREM 4. *For inputs sorted in reverse order, 2WRS generates one run containing all the input records.*

PROOF. The records obtained from the input are smaller than all the records in memory. However, in contrast to RS, those records are inserted in the BottomHeap. Since all the records from the BottomHeap can be used in the current run, all the stream is released in a single run through the BottomHeap. \square

THEOREM 5. *For inputs consisting of alternating chunks of length k records sorted in ascending order and k records sorted in descending order repeatedly, RS generates runs with an average length around twice the size of the memory m ($m \ll k$).*

PROOF. Let m be the size of the memory. Every chunk of k records sorted in ascending order is placed in the same run, as per Theorem 1.

When the algorithm starts reading records sorted in reverse order, only the first $m/2$ will be included in the current run. The rest of the records sorted in reverse order are put in runs of length m , as per Theorem 3. Therefore, the number of runs generated in a descending section is $\lfloor \frac{k - \frac{m}{2}}{m} \rfloor = \lfloor \frac{k}{m} - \frac{1}{2} \rfloor$.

The last m records of a chunk of k records sorted in reverse order $((k - \frac{m}{2}) \bmod m)$ are placed in the same run as the following k records sorted in ascending order.

So every chunk of k records sorted in ascending order is included in a run together with $m/2$ records from the next chunk of k records sorted in descending order, plus the last records from the previous run $(k - \frac{m}{2}) \bmod m$.

The average run length is then the total number of records divided by the number of generated runs,

$$\frac{2k}{1 + \lfloor \frac{k}{m} - \frac{1}{2} \rfloor} \quad (1)$$

The denominator of this formula can take the values $\lfloor \frac{k}{m} \rfloor$ and $\lfloor \frac{k}{m} + 1 \rfloor$. The formula achieves maximum value when the denominator is minimum. The maximum average run length is then

$$\frac{2k}{\lfloor \frac{k}{m} \rfloor} \approx 2m \quad (2)$$

\square

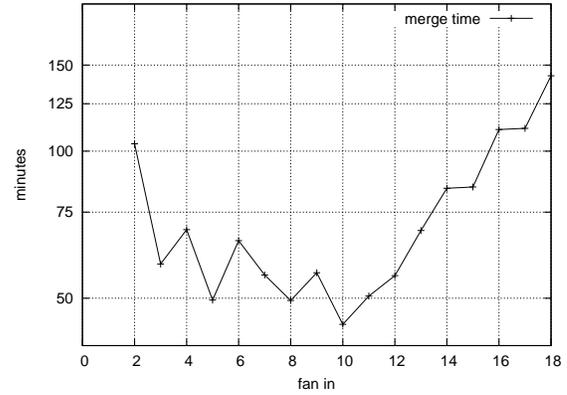


Figure 13: Merge time for different values of fan-in.

THEOREM 6. *For the inputs of Theorem 5, two-way replacement selection generates runs with an average length equal to k (with an appropriate heuristic³).*

PROOF. 2WRS behaves identically to RS for the chunks sorted sorted in ascending order, thanks to the TopHeap. For the chunks with records sorted in reverse order, 2WRS captures the trend with the BottomHeap, generating runs of k records, as well. Thus, the average run length is k . \square

E. FAN-IN ANALYSIS

In this experiment, we measure the fan in that achieves the best performance in our computer. In our experiment, we generate 400 files, each one with size 16MB, which contain integers already sorted following a uniform distribution (i.e. 400 runs), and we merge them. This experiment is independent of the algorithm that generates the runs, thus, it is valid for RS and 2WRS.

The fan in is a compromise between two characteristics: (a) the smaller the fan in, the more sequential is the access to the files from disk, but (b) the larger the fan in, the less merge operations are required to end the task.

We observe this tradeoff between the two benefits in Figure 13. If the fan in is too small, the algorithm takes more time because it must perform more merge steps. However, if the fan in is too large, the head of the disk performs more seeks and the bandwidth obtained from the disk is smaller. In our experiments, the minimum time was observed for a fan in 10, which means that in each merge step 10 different files are simultaneously merged.

³An appropriate heuristic is one that uses the TopHeap for sorted inputs and the BottomHeap for reverse sorted inputs.